Exercise walk-through

13.1.1

Designing a user-defined type for a problem, and the linked list

A-Level Computer Science

You must be able to

- show understanding of why **user-defined types** are necessary
- define and use **non-composite types**, including **enumerated** and **pointer** types
- define and use **composite data types**, including **set**, **record** and **class/object**
- **choose and design an appropriate user-defined data type for a given problem**

1 The method

Method — Designing a type from the problem, in four steps

Do not start from the list of type kinds and look for somewhere to use them. Start from the problem and let the type kinds be the **answers**:

1. **One record per thing the program stores.** A student, a book, a booking.
2. **One field per fact about that thing.** Write the facts down in plain words first.
3. **Choose each field's type.** A built-in type if one fits (`STRING`, `INTEGER`, `REAL`, `BOOLEAN`, `DATE`); an **enumerated** type if the fact is exactly one of a fixed few named values; a **SET** if it can be several of a fixed few at once; **another record** if the fact has parts of its own.
4. **Declare the types the record needs before the record itself**, because a declaration may only use a type that already exists.

Method — Designing a type from the problem, in four steps

```
TYPE HouseType = (Alpha, Beta, Gamma, Delta)

TYPE DateType
    DECLARE Day : INTEGER
    DECLARE Month : INTEGER
    DECLARE Year : INTEGER
ENDTYPE
```

Method — Designing a type from the problem, in four steps

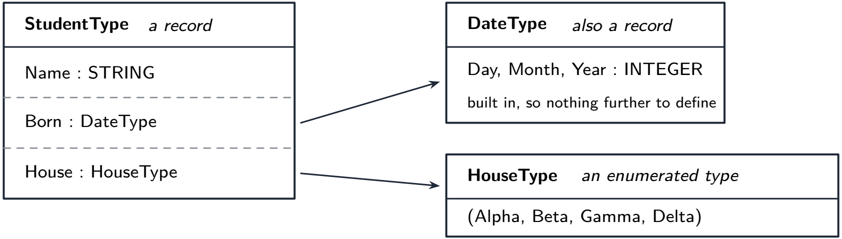
```
TYPE StudentType
    DECLARE Name : STRING
    DECLARE Born : DateType
    DECLARE House : HouseType
ENDTYPE
```

Method — Designing a type from the problem, in four steps

Reaching a nested field just extends the dot: `Class[3].Born.Year ← 2009`.
△ **"Why a user-defined type?" is not "because it is neater."** The marks are for: the compiler can then **check** that only a sensible value is stored (`House ← Alpha` is allowed, `House ← "Alphaa"` is not); one variable carries all the facts about one thing, so they cannot drift apart; and the type can be **passed, returned and stored in an array as a single item**.

Method — Designing a type from the problem, in four steps

Do not start from the list of type kinds and look for somewhere to use them.



Design from the *problem*, not from the type list: one **record** per thing the program stores, one **field** per fact about it, and then choose each field's type – built in where one fits, an **enumerated** type where the fact is one of a fixed few, and *another record* where the fact has parts of its own.

Method — The pair of declarations that builds a linked list

A node is a **record** holding the data plus a **pointer** to the next node, and the two types name each other. Declare the **pointer type first**: it may name a record type not yet defined, but the record cannot name a pointer type that does not exist.

Method — The pair of declarations that builds a linked list

```
TYPE PNode = ^NodeType

TYPE NodeType
    DECLARE Value : INTEGER
    DECLARE NextPtr : PNode
ENDTYPE

DECLARE HeadPointer : PNode
HeadPointer ← NULL
```

Method — The pair of declarations that builds a linked list

- **Insert at the front** – point the new node at the old head, *then* move the head. Doing those two in the other order loses the rest of the list:

```
DECLARE NewNode : PNode
NewNode ← NEW NodeType
NewNode^.Value ← 12
NewNode^.NextPtr ← HeadPointer
HeadPointer ← NewNode
```

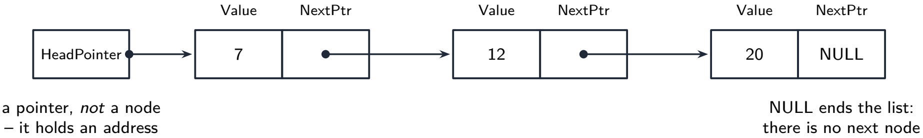
- **Traverse** – start at the head, follow `NextPtr`, stop at `NULL`:

Method — The pair of declarations that builds a linked list

```
DECLARE Current : PNode
Current ← HeadPointer
WHILE Current <> NULL DO
    OUTPUT Current^.Value
    Current ← Current^.NextPtr
ENDWHILE
```

Method — The pair of declarations that builds a linked list

A node is a record holding the data plus a pointer to the next node, and the two types name each other.



The node record and the pointer type each name the other, so the **pointer type is declared first** – `TYPE PNode = ^NodeType` above `TYPE NodeType`. To walk the list, follow `NextPtr` from the head and stop when it is `NULL`; to insert at the front, point the new node at the old head and then move the head.

Three nodes, each holding a value and a pointer, with a head pointer and NULL at the end

Method — Array of records, or linked list?

	array of records	**linked list**
size	fixed when declared	grows and shrinks while running
finding item n	direct, one step	follow n pointers from the head
inserting in the middle	shuffle every later item along	change two pointers
memory	one block, some of it unused	one node at a time, plus a pointer each

Choose by what the problem does most: if it mostly **looks items up by position**, an array; if it mostly **inserts and deletes**, or the number of items is unknown, a linked list.

2 Practice

Practice 2.1 ▪ 3 marks

A program stores, for each member of a sports club, their name, their date of birth, and which one of the four teams Red, Blue, Green or Yellow they belong to. State the user-defined type you would use for the date and the one you would use for the team, and justify each choice.

Solution 2.1

Method: Designing a type from the problem, in four steps — p.4

Worked steps

- for the date, a **record** (a user-defined composite type) **B1**
- because a date has parts of its own – a day, a month and a year – which belong together in one field **B1**
- for the team, an **enumerated** type, because the value is exactly one of a fixed, known list of names, and declaring it that way lets the compiler reject anything else **B1**

Practice 2.2 ▪ 3 marks

Declare a record type `DateType` with three integer fields for the day, month and year.

Solution 2.2

Method: Designing a type from the problem, in four steps — p.4

Worked steps

Solution 2.2

```
TYPE DateType
    DECLARE Day : INTEGER
    DECLARE Month : INTEGER
    DECLARE Year : INTEGER
ENDTYPE
```

- `TYPE DateType ... ENDTYPE` **B1**
- three `DECLARE` lines inside it **B1**
- all three of type `INTEGER` **B1**

Practice 2.3 ▪ 3 marks

Declare a record type `MemberType` whose fields are a name, a `DateType` and an enumerated type `TeamType`. You may assume `TeamType` has already been declared.

Solution 2.3

> Method: Designing a type from the problem, in four steps — p.4

Worked steps

Solution 2.3

```
TYPE MemberType
    DECLARE Name : STRING
    DECLARE Born : DateType
    DECLARE Team : TeamType
ENDTYPE
```

- `TYPE MemberType … ENDTYPE` **B1**
- a `STRING` field for the name **B1**
- fields of type `DateType` and `TeamType` **B1**

Practice 2.4 ▪ 2 marks

State **two** advantages of a linked list over an array of records.

Solution 2.4

Method: Array of records, or linked list? — p.14

Worked steps

any **two** of: **B1** *each*

- it can grow and shrink while the program runs, so the number of items need not be known when it is declared
- inserting or deleting in the middle changes only two pointers, instead of shuffling every later item along
- memory is used one node at a time, so none is reserved for items that are never stored

Practice 2.5 ▪ 1 mark

State what the `NextPtr` field of the **last** node in a linked list holds.

Solution 2.5

Method: The pair of declarations that builds a linked list — p.9

Worked steps

- `NULL` – there is no next node **B1**

3 Exam-style

Exam-style 3.1 ▪ 1 mark

Which of these is a **non-composite** user-defined type?

A record

B set

C enumerated

D class

Solution 3.1

Method: Designing a type from the problem, in four steps — p.4

A record

B set

C enumerated ✓

D class

Worked steps

C

- an enumerated type is a list of single named values, so it is **non-composite** **B1**
- a record, a set and a class each group several values together, so all three are composite

Exam-style 3.2 ▪ 7 marks

A library program stores, for each book, its title, its author, the year it was published, and whether it is currently on the shelf, on loan or lost.

Exam-style 3.2 (a) ▪ 2 marks

Declare a suitable **enumerated** type `StatusType` for the last of these.

Solution 3.2 (a)

Method: Designing a type from the problem, in four steps — p.4

Worked steps

```
TYPE StatusType = (OnShelf, OnLoan, Lost)
```

Solution 3.2 (a)

- `TYPE StatusType = ( … )` **B1**
- exactly the three values, and no others **B1**

Exam-style 3.2 (b) ▪ 3 marks

Declare a record type `BookType` for one book, using `StatusType`.

Solution 3.2 (b)

Worked steps

```
TYPE BookType
    DECLARE Title : STRING
    DECLARE Author : STRING
    DECLARE YearPublished : INTEGER
    DECLARE Status : StatusType
ENDTYPE
```

Solution 3.2 (b)

- `TYPE BookType ... ENDTYPE` B1
- two `STRING` fields and an `INTEGER` field B1
- a field of type `StatusType` B1

Exam-style 3.2 (c) ▪ 2 marks

Declare an array `Catalogue` able to hold 500 books, and write the statement that sets the status of the first book in the array to on loan.

Solution 3.2 (c)

Worked steps

```
DECLARE Catalogue : ARRAY[1:500] OF BookType
Catalogue[1].Status ← OnLoan
```

- the array declaration, of 500 elements of `BookType` **B1**
- the assignment, reaching the field with a dot and using the enumerated value unquoted **B1**

Exam-style 3.3 ▪ 10 marks

A program stores a list of integers in a linked list. Each node holds one integer and a pointer to the next node.

Exam-style 3.3 (a) ▪ 4 marks

Declare the pointer type `PNode` and the record type `NodeType`, in the correct order.

Solution 3.3 (a)

> Method: The pair of declarations that builds a linked list — p.9

Worked steps

Solution 3.3 (a)

```
TYPE PNode = ^NodeType

TYPE NodeType
    DECLARE Value : INTEGER
    DECLARE NextPtr : PNode
ENDTYPE
```

Solution 3.3 (a)

- `TYPE PNode = ^NodeType` **B1**
- the pointer type declared **first**, because the record needs a type that already exists **B1**
- `TYPE NodeType` … `ENDTYPE` with an `INTEGER` field **B1**
- a `NextPtr` field of type `PNode` **B1**

Exam-style 3.3 (b) ▪ 3 marks

The list is empty, and then the values 20, 12 and 7 are each inserted **at the front**, in that order. State the value in the node that `HeadPointer` points to afterwards, and state what the `NextPtr` of the node holding 20 contains.

Solution 3.3 (b)

Worked steps

- inserting at the front makes the **most recent** value the first one, so `HeadPointer` points to the node holding **7** **B1**
- 20 was inserted first, so nothing was ever attached after it **B1**
- its `NextPtr` therefore still holds **`NULL`**, and it is the last node in the list **B1**

Exam-style 3.3 (c) ▪ 3 marks

Write pseudocode that outputs every value in the list, starting from `HeadPointer`.

Solution 3.3 (c)

Worked steps

```
DECLARE Current : PNode
Current ← HeadPointer
WHILE Current <> NULL DO
    OUTPUT Current^.Value
    Current ← Current^.NextPtr
ENDWHILE
```

Solution 3.3 (c)

- start from the head: `Current ← HeadPointer` **B1**
- a loop that continues while `Current <> NULL` **B1**
- output the value and then move on with `Current ← Current^.NextPtr`, in that order **B1**
- △ moving on before outputting loses the first value; testing `Current^.NextPtr <> NULL` instead loses the last one